Ray Shooting in Logarithmic Time

Tibor Fóris, Gábor Márton, László Szirmay-Kalos
Department of Process Control
Technical University of Budapest
Budapest, Műegyetem rkp. 9/R, H-1111, Hungary
foris@fsz.bme.hu, marton@fsz.bme.hu

Abstract

A new technique for ray tracing acceleration is presented. The objects of the scene are preprocessed to form a data structure that makes possible to find the first intersection with an arbitrary ray at the speed of binary search, that is in time logarithmically proportional to the number of objects. The idea is based on a peculiar representation of rays, while the underlying data structures are spherical and planar subdivisions, balanced binary trees.

Keywords: computational geometry, ray shooting, ray tracing.

1 Introduction

Ray tracing [1] is still considered as one of the most promising and at the same time the slowest image synthesis method. Its speed is highly affected by the query time of the *intersection oracle* that returns the first intersection along a given ray. Reducing this query time is permanently subject of thorough research.

The problem is known as the *ray shooting problem* in computational geometry: *given a set of objects in 3-space, preprocess them into a data structure so that the first object intersected by an arbitrary query ray can be retrieved efficiently.* Mark de Berg has recently developed efficient ray shooting algorithms [2, 3]. His most general algorithm can shoot arbitrary rays into a set of arbitrary polyhedra with n edges altogether, with a query time of $O(\log n)$ and preprocessing time and storage of $O(n^{4+\varepsilon})$, where ε is an arbitrarily small positive constant. Other algorithms has also been published recently [4, 5]. Implementations of these algorithms, however, have not been published yet; it is no wonder since they are usually based on sophisticated constructs of computational geometry, such as hyperplane arrangements and cuttings, which are themselves fairly complicated. One important feature of these ray shooting algorithms is that they aim at *worst-case optimization.* The other important feature is that the logarithmic query time is reached by different generalizations of the idea behind binary search: use a data structure which makes it possible to exclude a fixed portion, say $(1/r)$th, of the objects from the set of possible answers in each consecutive search step in constant time, where r is a constant independent from the number of objects ($r = 2$ in the case of binary search).

The practice of computer graphics, on the other hand, produced its own techniques that can be called *heuristic methods.* Arvo and Kirk presented a comprehensive overview of the basic techniques [6]. One implortant feature of these algorithms is that they do not aim at worst-case optimization but rather aim at *average-case optimization* and are much easier to implement than the recently emerged computational geometric methods.

The worst case query time of any of these algorithms is $O(n)$ (like that of brute-force search), but the average-case query time is $O(1)$ if the objects are evenly distributed, as it was shown at the previous Winter School for the case of the regular grid method [7] by using a technique that is applicable to the other methods as well [8]. The other important feature is that this excellent average-case query time is reached by an inherent common characteristic of these algorithms: they test the candidate objects for intersection (roughly) in the order of their distance from the origin of the ray.

We would like to introduce a new technique for ray shooting, which is closer to computational geometric algorithms than to the heuristic ones in the sense that it aims logarithmic worst-case query time. The algorithm, that is based on a peculiar representation of rays and uses data structures such as spherical and planar subdivisions and balanced binary trees, has also been implemented in two dimensions.

2 The idea

A ray in the d-dimensional space ($d = 2, 3$ in the practically important cases) can be represented by $2d - 1$ independent parameters: d of them can describe its origin, while $d - 1$ of them can describe its direction (polar coordinates). There is, however, another possible distribution of these independent parameters: $d-1$ of them henceforward describe the direction, but the remaining d number of them is distributed differently. Let the plane containing the origin of the coordinate system and having a normal vector parallel to the direction of the ray be called the *complementer plane* of the ray, as Figure 1 shows. Let furthermore the orthogonal projection of the ray onto this plane be called the *supporting point* of the ray. Then the next $d - 1$ of the ray parameters (b in the figure) define the position of the supporting point on the complementer plane, while the last one (c in the figure) is the signed direction of the ray origin from the supporting point.

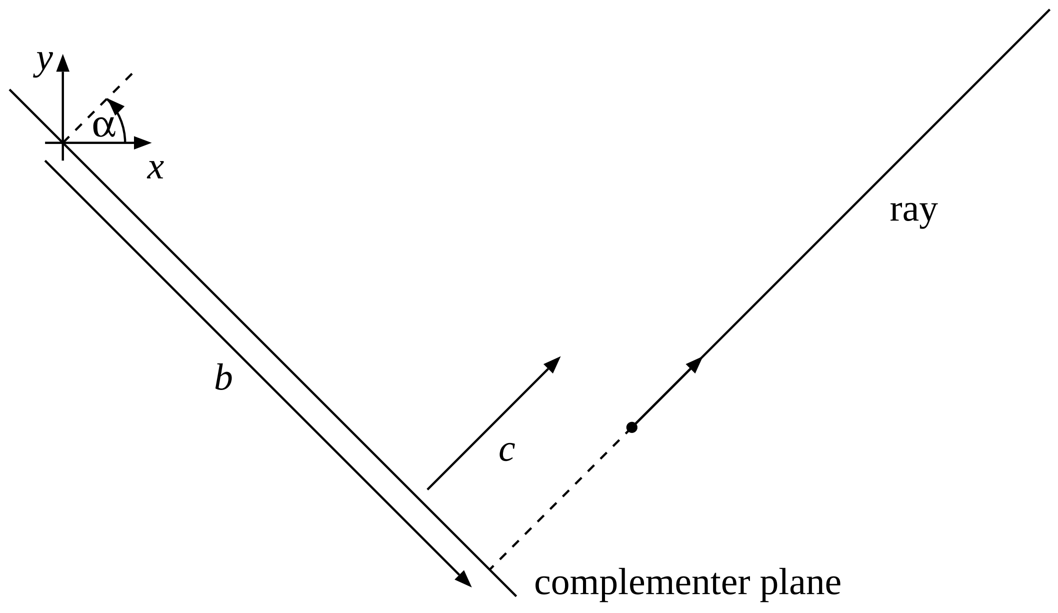

Figure 1: Representation of the ray

But what can be expected from this kind of representation of the ray? Since *point location* can be efficiently performed in subdivisions, the problem of ray shooting is tried to be traced back to point location problems, or more precisely, to a series of point

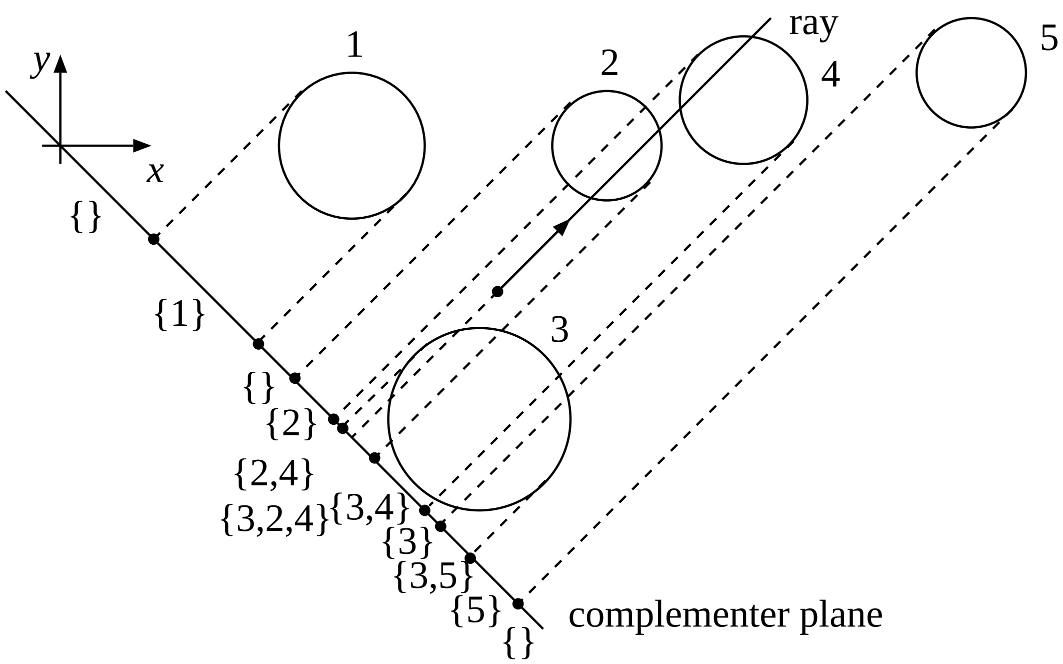

Figure 2: Representation of objects on a complementer plane

location problems. On the complementer plane of any ray, the objects of the scene can be represented by first computing the subdivision of this plane induced by the orthogonal projection of the objects and then assigning an ordered list of objects to each of the *homogeneous* territories of this subdivision, as Figure 2 shows. The ordering of the lists should be in accordance with the spatial relation of the objects. If the objects are simple and disjoint then this *projection diagram* is relatively simple to compute, while in the case of non-disjoint or complicated objects it is getting harder to construct the subdivision. If this projection diagram corresponding to the query ray were always within reach, then we would take it, locate the supporting point of the ray in the diagram, that is determine the territory containing the point, take the ordered list corresponding to this territory, locate the ray origin in it and finally take the next object from the list. Both point location in a planar subdivision [9] and searching in an ordered list can be performed in logarithmic time, hence our ray shooting algorithm would possess logarithmic query time. Note that although the idea is best suited to convex objects, it is feasable for non-convex objects as well.

Only one problem is unsolved: the number of different complementer planes is infinite and hence the possible projection diagrams cannot be stored in advance. Fortunately, however, there is only a finite number of *topologically different* projection diagrams. The topologically equivalent diagrams induce a subdivision of the space of ray directions (that is the surface of the d-dimensional unit sphere): a *connected* territory on the surface of the *direction sphere* corresponds to an equivalence class of topologically equal and only metrically different projection diagrams. Finding the territory containing the direction of the ray in this directional subdivision is also a point location problem that can be solved efficiently. The only remaining difficulty is that the preprocessing and searching algorithms associated with the *metrically unevaluated* projection diagrams should be able to work in a parameterized-at-run-time fashion, since such a diagram corresponds to not only one but lots of rays.

Only the two-dimensional case will be overviewed in the rest of the article. The ideas are more or less extendable to three dimensions.

3 The two-dimensional case

For the sake of simplicity, let the object scene consist of *disjoint spheres*. Extension to non-disjoint spheres or complicated objects is possible but not detailed here.

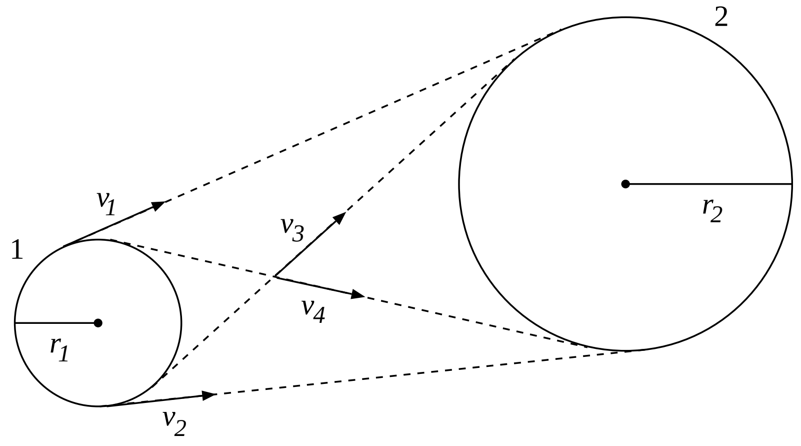

Figure 3: Directions where topology changes

The first problem is the computation of the directional subdivision. The direction circle should be subdivided into regions corresponding to topologically equivalent projection diagrams. Imagine that we are moving on the contour of the direcion circle and keep our eyes on the actual projection diagram. We can notice that the structure of the diagram changes in discrete steps: the direction position where such a change occurs is the direction of one of the inner or outer common tangent lines associated with a pair of objects, as Figure 3 shows (v_1, v_2, v_3, v_4). Thus the directional subdivision is determined by the directions of all the possible common tangent lines between the objects.

To each of the resulting directional intervals a different projection diagram should be associated, as Figure 4 shows for the case of two objects. The figure also demostrates that the method requires tremendous storage: even for the case of two objects, eight complete projection diagram is stored and this number increases proportionally to the square of the number of objects. The storage requirement, however, can be reduced by exploiting *coherence* of projection diagrams. First of all not that the whole data structure is symmetric: the projection diagram corresponding to a direction v is essentially the same as the one corresponding to the opposite direction $-v$, only the ordering of the lists are opposite; exploiting this property halves storage requirement.

The other manifestation of coherence of projection diagrams is that the closer to each other two such diagrams are in the directional subdivision, the less they differ; two adjacent projection diagrams differ in one and only one list, the rest of the lists are the same. One possibility to exploit this would be if the ordered lists were stored not separately but each possible list were stored only once (centrally) and projection diagrams had only references to the corresponding lists. This technique saves storage again but does not reduce complexity. The reduction of storage complexity should involve a more sophisticated data structure which makes it possible to construct the final form of the

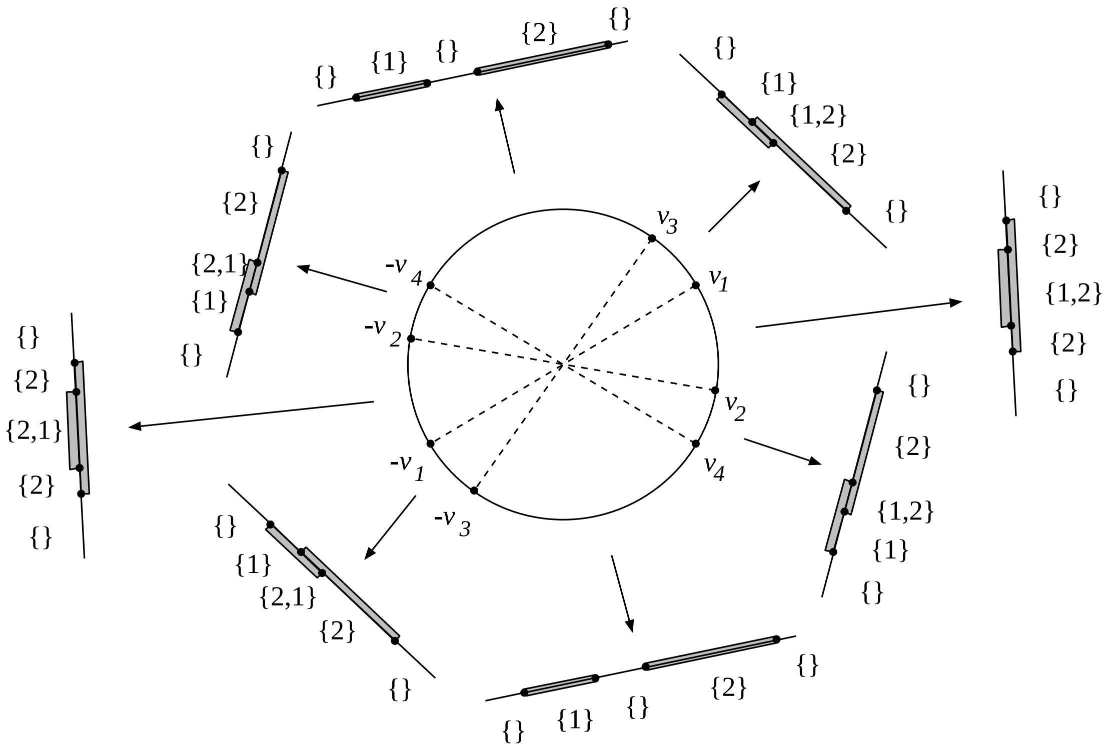

Figure 4: Different projection diagrams

projection diagram corresponding to the direction of the actual query ray in search time (at low cost), but this is left for future research.

4 Implementation details

The two-dimensional version of the algorithm has been implemented in `C++` using object oriented design technique. The diagram of the relevant classes is shown in Figure 5 using the notations of the Rumbaugh methodology [10].

`SearchStructure` is built incrementally in the preprocessing phase, that is, each consecutive object is inserted into a complete, valid structure of the previous ones. It realizes the directional subdivision and contains the `ProjectionDiagram`s. Containment is implemented by a balanced binary tree in order to achieve logarithmic search time. `ProjectionDiagram` contains the homogenous `ProjectionTerritory`s. This containment is also implemented by a balanced binary tree. A `ProjectionTerritory` is defined by its begin and end point, each of which being either the point where the projection of an object starts (`BeginObject`) or the point where the projection of an object ends (`EndObject`). The homogenous `ProjectionTerritory` also contains the corresponding list of (projecting) objects, `ObjectList`, again a balanced binary tree.

Once `SearchStructure` has been built, its method `Shoot` can be used to retrieve the first object intersected by the query ray. First the ray parameters α, b, c are computed and then they are used to parameterize the methods of the contained objects. Parameter α is first computed: it is the angle between the ray direction and the x axis, according to Figure 1. Parameters b, c are then computed according to the following transformation:

$$\left[\begin{array}{cc} b & c \end{array}\right] = \left[\begin{array}{cc} x & y \end{array}\right] \left[\begin{array}{cc} \sin\alpha & \cos\alpha \\ -\cos\alpha & \sin\alpha \end{array}\right],$$

Figure 5: Object diagram

where x, y are the (world) coordinates of the ray origin and α is the previously computed angle of the ray direction. Note that α, b, c appear as `alpha`, `b`, `c` in Figure 5.

5 Complexity

Let n denote the number of objects. Note first of all that if the size of the involved subdivisions (directional subdivision, projection diagrams, ordered object lists) is polinomial in n, then the query time is logarithmic, since $O(\log n^{k_1} + \ldots + \log n^{k_m}) = O(\log n)$. This is definitely true in our case.

Since each pair of objects induces 8 (4) new division point on the direction circle (on the half of the direction circle), the number of projection diagrams is $O(n^2)$. The size of one projection diagram is $O(n^2)$ hence the storage requirement is $O(n^4)$.

Inserting a new object into the structure involves the the following activities for each pair formed by this new object and an already inserted one:

- insert the 8 (4) new division points into the directional subdivision — this requires $O(\log n)$ time;
- make a copy of the projection diagrams corresponding to the 8 (4) "disturbed" regions of the directional subdivision — it is performed in $O(n^2)$ time.

Counting each pair, it takes $O(n^3)$ time altogether. Then the new object is inserted into the $O(n^2)$ number of existing projection diagrams in $O(n \log n)$ time per each. Thus inserting one object lasts $O(n^3 \log n)$ time. The total preprocessing time is $O(n^4 \log n)$.

6 Summary

A data structure involving directional subdivision and projection diagrams has been proposed for solving the ray shooting problem efficiently. Having the data structure built, the first intersection along an arbitrary query ray is possible to find in logarithmic time in the worst case. The price of this quick search is the size of the data structure and the preprocessing time. Future research work may be devoted to reduce storage complexity and preprocessing time using more sophisticated data structures.

7 References

[1] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[2] Mark de Berg. *Ray shooting, depth orders and hidden surface removal.* Springer-Verlag, Berlin, Germany, 1993. x+201 pp.

[3] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12(1):30–53, 1994.

[4] R. Bar-Yehuda and S. Fogel. Variations on ray shooting. *Algorithmica*, 11(2):133–45, 1994.

[5] P.K. Agarwal and J. Matousek. Ray shooting and parametric search. *SIAM Journal on Computing*, 22(4):794–806, 1993.

[6] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201–262. Academic Press, London, 1989.

[7] Gábor Márton and László Szirmay-Kalos. On average-case complexity of ray tracing algorithms. In *Winter School of Computer Graphics '95*, pages 187–196, Plzen, Czech Republic, 14–18 February 1995. University of West Bohemia.

[8] Gábor Márton. *Stochastic Analysis of Ray Tracing Algorithms.* PhD thesis, Department of Process Control, Technical University of Budapest, Budapest, Hungary, 1995. prior to public debate, in Hungarian.

[9] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[10] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Inc., Englewood Cliffs, New Yersey 07632, 1991.